Université Claude Bernard Lyon 1

IA5 - Intelligence Bio-Inspirée

TP2 - Apprentissage profond par renforcement

*V*alentin CHAFFRAIX *J*ulien VERMOREL

6 janvier 2020

Table des matières

1 Réplicabilité et code source

Les sources de notre projet sont disponibles sur un dépot github à cette adresse : `https://github.com/vchaffraix/ibi-drl`.

Nous avons effectué nos expérimentations avec des scripts Python3.7 et PyTorch dans un environnement virtuel. La procédure de réplicabilité est détaillée dans le fichier `README.md` et les dépendances dans le fichier `requirements.txt`.

2 Deep Q-network sur CartPole

2.1 Début

2.1.1 Question 1 - Agent aléatoire

Le code de l'agent CartPole est disponible dans le fichier source `CartPole-v1.py`, on peut observer le fonctionnement de l'agent aléatoire dans la classe `RandomAgent`.

2.1.2 Question 2 - Évolution de la récompense

Nous avons choisi de visualiser l'évolution de la récompense à l'aide de `matplotlib` en affichant la récompense cumulée par épisode ce qui nous permet de visualiser globalement comment l'agent a performé sur un épisode.

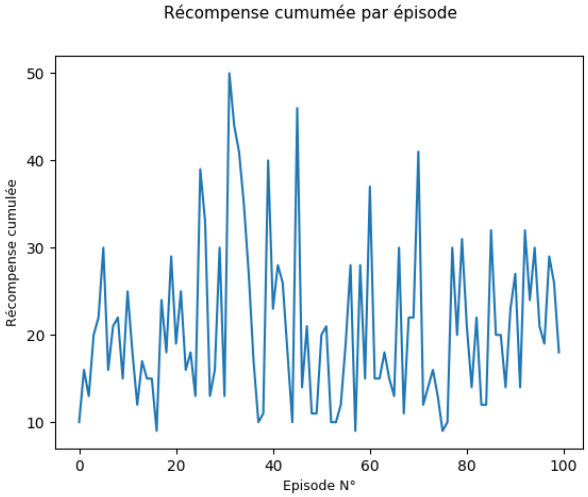

FIGURE 1 – Exemple pour 100 épisodes avec l'agent aléatoire

On voit bien que les résultats sont totalement chaotiques et on n'observe pas d'apprentissage.

2.2 Experience replay

2.2.1 Question 3 - Interaction

La structure de données d'une interaction est visible dans la classe `Interaction` qui est équivalente à un simple tuple (e, a, s, r, f).

2.2.2 Question 4 - Buffer

Le code du buffer est disponible dans la classe `Buffer` qui possède les méthodes `.append()` et `.sample()` pour pouvoir gérer l'experience replay.

2.3 Deep Q-learning

2.3.1 Question 5 - Réseau de neurones

Le code du réseau est disponible dans la classe `QModel`, le réseau est structuré selon les couches suivantes :

— La couche d'entrée de taille 4 (état de 4 dimensions)
— Une couche *fully connected* de 32 neurones
— Une couche *fully connected* de 32 neurones
— Une couche *fully connected* de 32 neurones
— La couche de sortie de 2 neurones (2 actions possibles)

2.3.2 Question 6 - Exploration et curiosité

Nous avons implémenté la statégie d'exploration de *Boltzmann* ainsi que la méthode ϵ-*greedy*, le code se trouve dans la méthode `act` de la classe `DQN_Agent`. Nous avons également choisi de faire diminuer la valeur de ϵ ou de τ pendant l'apprentissage pour d'abord commencer avec des actions aléatoire puis pour se diriger vers les meilleures actions.

2.3.3 Question 7 - Apprentissage

Nous avons choisi de calculer de calculer l'erreur sur l'ensemble du *minibatch* en une seule passe sur le réseau, cela a déjà pour effet d'accélérer le temps de calcul en utilisant le calcul tensoriel de PyTorch mais cela nous permet aussi d'éventuellement de converger plus vite vers des minimums locaux.
Le code de l'algorithme se trouve dans la méthode `learn` de `DQN_Agent`.

2.3.4 Question 8 -Target Network

Nous avons implémenté les deux approches de mise à jour avec les hyperparamètres α pour l'approche de *Polyak* et la fréquence de recopiage pour l'approche par copie. Le code de la mise à jour du *Target Network* est disponible dans la méthode `learn` de `DQN_Agent`.

2.4 Résultats

Après plusieurs modifications des hyperparamètres nous avons réussi à obtenir des résultats satisfaisant dans l'environnement `CartPole-V1`.

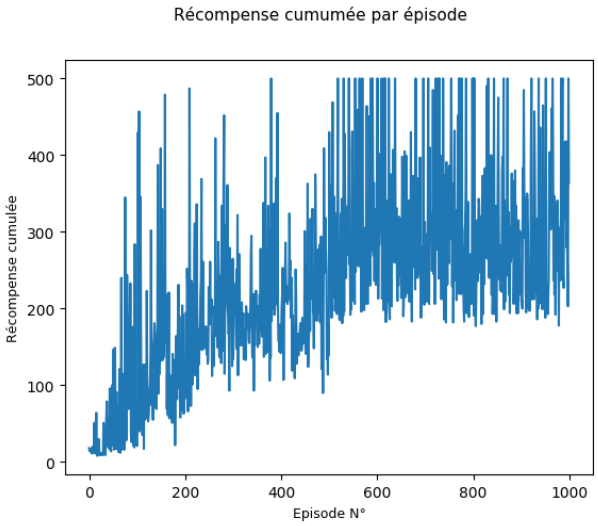

FIGURE 2 – Résultats sur 500 épisodes d'entrainement et 500 épisodes de test

On peut observer que durant la phase d'exploitation de la politique apprise, notre agent arrive plusieurs fois à atteindre un score de 500, ce qui est le maximum possible dans l'environnement `CartPole-V1`. Nous avons obtenu ces résultats avec les hyperparamètres suivant :

Hyperparamètre	**Valeur**
ϵ_{init}	1
ϵ_{decay}	0.99
ϵ_{final}	0.1
γ	0.9
η	0.001
Taille minibatch	100
Stratégie d'exploration	ϵ-greedy
MAJ du *Target Network*	Recopiage
Fréquence de mise à jour	1000

FIGURE 3 – Hyperparamètres utilisés

Malheureusement nous n'avons pas implémenté la sauvegarde du modèle sur CartPole, ainsi pour pouvoir reproduire ces résultats il sera nécessaire de relancer les 500 épisodes d'entraînement.

3 Environnement Breakout Atari

Le code pour l'environnement Breakeout est disponible dans le fichier `BreakoutNoFrameskip-V4.py`.

3.1 Question 1 - Preprocessing

Nous avons d'abord implémenté le preprocessing manuellement mais nous nous sommes rendu compte que tout était déjà disponible à l'aide des wrappers `wrappers.AtariPreprocessing` et `wrappers.FrameStack`. On peut voir leur utilisation dans le constructeur.

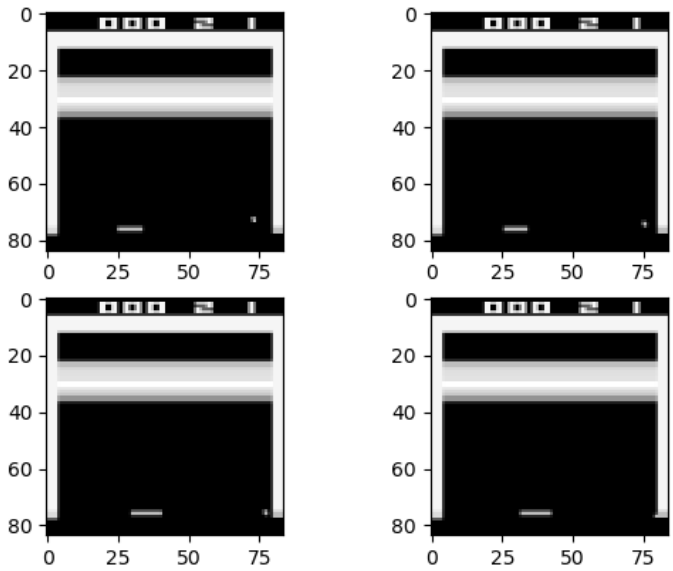

FIGURE 4 – Représentation d'un état

On a bien 4 frames en niveau de gris de taille 84x84 qui fait un état de dimension 3, 4x84x84.

3.2 Question 2 - Adaptation du code

Nous avons rajouté beaucoup de chose au code pour pouvoir notamment sauvegarder le modèle après entraînement à cause des longs temps d'exécution. Tous les hyperparamètres sont modifiables dans un dictionnaire python `PARAMS` qui se trouve dans le main.

3.3 Question 3 - Réseau de neurone convolutionnel

Nous avons repris la structure du réseau utilisé par l'équipe de [Mnih et al., 2015] [1] :

- `Conv2d(4, 32, kernel_size=8, stride=4)`
- `Conv2d(32, 64, kernel_size=4, stride=3)`
- `Conv2d(64, 64, kernel_size=3, stride=1)`
- `Linear(fc_input_size, 512)`
- `Linear(512, a_size)`

1. [Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human- level control through deep reinforcement learning. Nature, 518(7540) :529.

La variable `fc_input_size` est calculée à l'exécution dans la fonction `conv2d_size_out` ce qui permet de facilement modifier la structure. La variable `a_size` vaut 4 dans cet environnment.

3.4 Question 4 - Hyperparamètres

Nous avons repris la plupart des hyperparamètres de l'article :

Hyperparamètre	**Valeur**
γ	0.9
Stratégie d'exploration	ϵ-greedy
ϵ_{init}	1
ϵ_{decay}	0.999
ϵ_{final}	0.1
η	0.001
m	4
`frame_skip`	4
Taille du buffer	10000
Taille minibatch	32
MAJ du *Target Network*	Recopiage
Fréquence de mise à jour	1000

FIGURE 5 – Hyperparamètres utilisés

Nous avons diminué la taille du buffer à 10000 car nous n'avions pas suffisament de mémoire pour stocker tous les états. En effet quand on sait qu'un état stocke deux fois 4 frames on se rend compte que la mémoire monte très vite. Quand *CUDA* est activé nous avons choisi de quand même stocker les frames dans la RAM plutôt que la VRAM car plus abondante malgré le te fait que les faire passer d'une mémoire à l'autre prend un petit peu de temps.

3.5 Question 5 - Comportement appris

Malgré nos nombreux essais nous n'avons malheurement pas réussi à faire apprendre notre agent. Nous avons testé avec plusieurs hyperparamètres mais l'agent choisi presque systématiquement de se placer sur un côté car cela lui permet de renvoyer la balle à sa première apparition. Pourtant nous avons considérablement descendu la vitesse de décroissance de ϵ pour favoriser l'aléatoire pendant plus longtemps pour forcer l'agent à essayer plus de choses. Mais il semble que l'agent n'est pas capable de "voir" la balle et cherche juste l'action qui va maximiser sa récompense. Malgré ça nous nous sommes dit que c'était simplement un manque d'apprentissage alors nous avons lancé une longue session d'apprentissage.

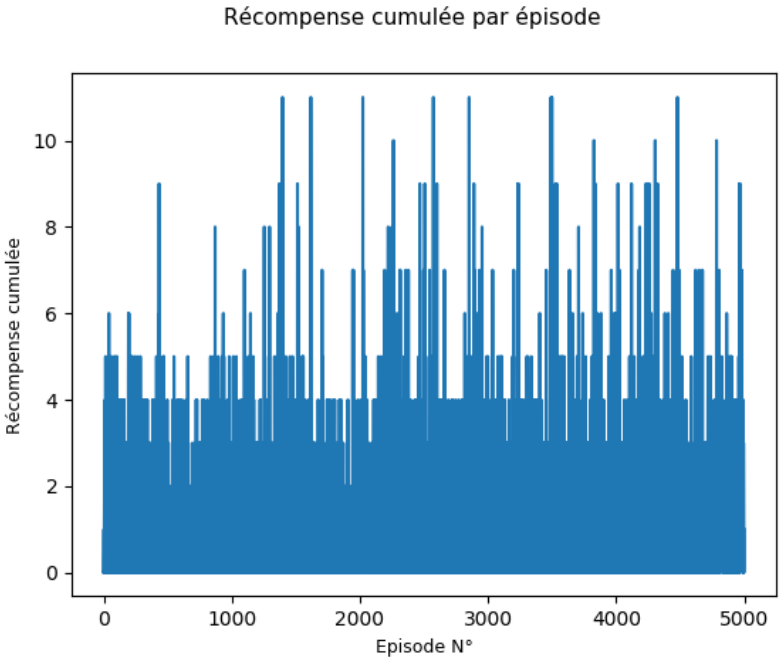

FIGURE 6 – Récompenses sur 5000 épisodes (13h d'entraînement)

Pourtant malgré tout on ne constate toujours aucun progrès.

Le modèle entraîné de cette session est disponible dans le dossier `model` du dépot github et la vidéo d'un épisode est disponible dans le dossier `video`.